MC^2: Multiple Clients on a Multilevel Cache

Gala Yadgar[1]*, Michael Factor[2], Kai Li[3] and Assaf Schuster[1]
[1] Computer Science Deptartment
Technion Israel Institute of Technology
Haifa 32000, Israel
{gala,assaf}@cs.technion.ac.il

[2] IBM Haifa Labs
Haifa University Campus
Haifa 31905, Israel
factor@il.ibm.com

[3]Department of Computer Science
Princeton University
Princeton, NJ 08540, USA
li@cs.princeton.edu

Abstract

In today's networked storage environment, it is common to have a hierarchy of caches where the lower levels of the hierarchy are accessed by multiple clients. This sharing can have both positive or negative effects. While data fetched by one client can be used by another client without incurring additional delays, clients competing for cache buffers can evict each other's blocks and interfere with exclusive caching schemes.

Our algorithm, MC^2, combines local, per client management with a global, system-wide, scheme, to emphasize the positive effects of sharing and reduce the negative ones. The local scheme uses readily available information about the client's future access profile to save the most valuable blocks, and to choose the best replacement policy for them. The global scheme uses the same information to divide the shared cache space between clients, and to manage this space. Exclusive caching is maintained for non-shared data and is disabled when sharing is identified. Our simulation results show that the combined algorithm significantly reduces the overall I/O response times of the system.

1. Introduction

Caching is used in storage systems to provide fast access to recently or frequently accessed data, reducing I/O delays and improving system performance. In many storage system configurations, client and server caches form a two- or more layer hierarchy. With a single client, the effectiveness of such hierarchies is optimal when caches are kept *exclusive*; a data block should be cached in at most one cache level at a time, avoiding data redundancy [29].

When the lower levels of the hierarchy are accessed by several clients, data sharing may occur. Sharing introduces both positive and negative effects on the performance of a multilevel cache. On the positive side, blocks fetched by one client may later be requested by another client. The second client experiences the effect of prefetching, when blocks are present in the shared cache and do not have to be fetched synchronously.

On the negative side, sharing introduces two major challenges. The first is maintaining exclusivity: a block may be cached in the first level cache by one client and in the second level by another. Furthermore, exclusivity may deprive clients of the benefit of sharing described above, since blocks fetched by one client are not available for use by others. Several previous studies acknowledge this problem [29, 31], while others assume that clients access disjoint data sets [26, 28].

The second challenge is meeting the demands of competing clients for space in the shared cache, while minimizing their interference with each other. A common approach for allocation in shared caches is *partitioning*, where each client is allocated a portion of the cache buffers according to a global allocation scheme [14, 28]. This approach is problematic when blocks are accessed by several clients and might belong to more than one partition.

We present MC^2, an algorithm for managing multilevel shared caches. We show that it enhances the positive effects of sharing and reduces the negative ones, by choosing the right replacement policy. MC^2 relies on information provided by the application running on each client. The clients are expected to provide a division of accessed blocks into

*Gala Yadgar's work is supported in part by the Levi Eshkol scholarship from the Israel Ministry of Science.

1063-6927/08 $25.00 © 2008 IEEE
DOI 10.1109/ICDCS.2008.29

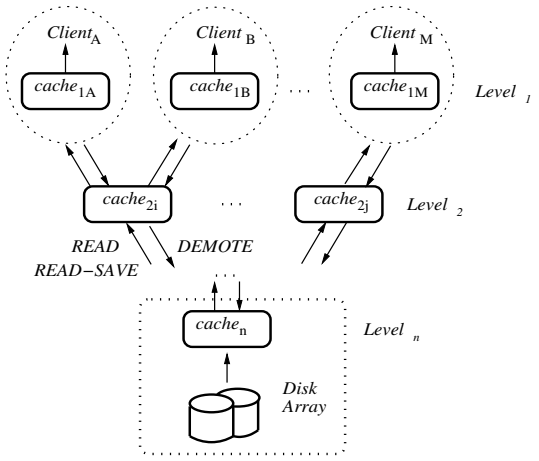


Figure 1. Our storage model consists of n cache levels, with possibly several caches in each level. READ and READ-SAVE fetch a block from level i to level $i-1$, DEMOTE sends a block from level $i-1$ to level i.

ranges and the frequency of access as well as the access pattern to each range. Previous studies have demonstrated that such information, in the form of application hints, is often available and is always useful for managing cache buffers [4, 5, 21, 24, 30].

The hints are used for two levels of management. On the local, per-client level, the hints determine which ranges are allocated space in the cache, and which replacement policy is most suitable for each range. On the global, system-wide level, the hints determine the replacement policy suitable for shared ranges and whether caching in each range is exclusive or inclusive. A global LRU-based scheme is used to divide the shared cache space between competing clients.

We examine our approach by simulating TPCH and TPCC workloads with a varying number of clients, comparing MC^2 to LRU, Demote [29], ARC [19], MultiQ [31] and an optimal lower bound on I/O response time [9]. Our results show that MC^2 reduces I/O response times below those of all other online policies, in almost all cache sizes, both for the single client setting and for multiple clients. MC^2 is the best policy to use in all sharing scenarios, with performance close to the optimal lower bound.

2. Storage model

Our model, shown in Figure 1, consists of several clients, each with its own cache. There are n cache levels, organized in a tree rooted at $cache_n$, which is attached to the storage level, $Disk$. The access cost of cache level i is C_i. The cost of a disk access is C_{Disk}. The cost of demoting a block from level $i-1$ to level i is D_i. We assume that a lower cache level carries an increased access cost, and that demoting and access costs are equal for a given level. Namely, $C_1 = D_1 < C_2 = D_2 < \ldots < C_n = D_n < C_{Disk}$.

Typically, the caches in the first level reside in the clients' memory and $cache_n$ resides on the storage controller. Additional cache levels may reside in either of these locations, as well as additional locations in the network. The access costs, C_i and D_i, represent a combination of computation, network, and queuing delays. C_{Disk} also includes seek times. The model is demand paging, read-only (for the purpose of this work, we assume a separately managed write cache), and defines three operations:

- READ (x, i) – move block x from $parent(cache)$ in level $i+1$ to $cache$ in level i, removing it from $parent(cache)$. If x is not found in $parent(cache)$, READ$(x, i+1)$ is performed recursively, stopping at $Disk$ if x is not found earlier.

- READ-SAVE (x, i) – copy block x from $parent(cache)$ in level $i+1$ to $cache$ in level i. If x is not found in $parent(cache)$, READ-SAVE $(x, i+1)$ is performed recursively, stopping at $Disk$ if x is not found earlier.

- DEMOTE (x, i) – move block x from $cache$ in level i to $parent(cache)$ in level $i+1$, removing it from $cache$.

Following previous studies [14, 28, 29], we assume no cooperation between caches in the same level. Each client may page blocks only from the cache directly attached to it, or its ancestors. Furthermore, we assume a client has no information about applications running on other clients.

Our goal is to minimize the *average I/O response time* for each client. While the load on the disk, the distribution of requests, and the access pattern all affect disk access costs, we focus on the performance of the cache hierarchy. Therefore, we assume a constant cost for all disk accesses. However, we note that when data sharing occurs, a cache miss does not always incur the same delay; when two clients fetch the same block, the second client issuing the request may experience a shorter response time than the first, as the request is already being processed.

While reducing I/O response times is our main goal, we wish to maintain a "fair" distribution of space between clients whenever possible. By "fairness" we will measure the performance improvement experienced by the clients, and not necessarily the amount of space allocated to them. We rely on a definition of fairness for processor multithreading [8] which we adapt to cache performance. We define the *speedup* of a client as the ratio between the completion time of its application when running alone and its completion time when running in a shared system. Fairness is defined as the minimum ratio between speedups in the system: $Fairness \equiv \frac{speedup_j}{speedup_k}$, where $Client_j$ is the client with minimal speedup in the system, and $Client_k$ is the one with maximal speedup. It follows from this definition that

$0 < Fairness \leq 1$. Perfect fairness is achieved when $Fairness = 1$, which means that all clients experience the same speedup (or slowdown).

Due to lack of space, we do not present the fairness maintained by all policies in our evaluation. However, Our experiments show that MC^2 is able to maintain fairness close to that of LRU, even though it does not always use LRU (or LRU-based policies) to manage the shared cache.

3. The MC^2 algorithm

Our algorithm, MC^2, consolidates the management of blocks that share the same access characteristics. For example, blocks from the same database file or index tree level belong to the same range, and are managed in their own cache partition. The size of each partition is dynamically adjusted, as explained below. The division of blocks into ranges, along with each range's size, access pattern, and the frequency of access to it, are supplied in the form of application hints by each client in the system. Each I/O request (in all cache levels) is tagged by the application with the range identifier of the requested block.

Whenever a new block is brought into the cache, and an old one must be evicted, the above information is used to answer three basic questions: Which client must give up a block? Which of its partitions must grow smaller? Which is the least valuable block in this partition?

In other words, the *allocation* aspect of MC^2 determines the quota of cache space for each range, and the *replacement* aspect manages blocks within each range. Each aspect is handled in two levels: *globally* (system-wide) and *locally* (per client). The global allocation scheme determines the quota of each client in the shared cache space, according to its frequency and efficiency of use of the cache. The local scheme determines the amount of space each range is allocated from its client's quota, according to its marginal gain. The local replacement scheme chooses the best replacement policy for each client's range, according to its hinted access pattern. The global replacement scheme chooses the best policy for shared ranges according to the combination of their access patterns.

3.1. Local allocation and replacement

We first describe how blocks are managed within the space allocated for each client. This space consists of the *client cache*, which is the first level cache used only by this client, and the space allocated for this client in shared caches in lower levels. We describe the global allocation and replacement schemes in the following subsections.

Within the space allocated for it, each client manages its blocks using Karma [30], an algorithm which provides exclusive allocation and replacement in a multilevel cache hierarchy with a single client. Note that the space allocated for each client can be viewed as such a hierarchy. We describe the main principles of the local management, and refer the reader to the description of Karma for further details.

Karma relies on the same application hints described above. These hints are used to compute the marginal gain for each range. Marginal gains induce an order of priority on all ranges – and thus on all blocks – in a trace. This order is used by each client to choose the ranges with highest priority and allocate a fixed partition for each one of them in its cache. Each range is managed separately, according to its access pattern. When a block is brought into the cache, a block from the same range is discarded, according to the range's policy. The best replacement policy is used for each access pattern: MRU for sequential and looping references, and LRU for random references.

Each cache level stores only blocks belonging to its assigned ranges. Blocks that will be saved in the cache are fetched using READ, while blocks that will be immediately discarded are fetched using READ-SAVE. All evicted blocks are sent to the lower level using DEMOTE, and the lower level saves them if space is allocated for their range. When a range is split between two cache levels, exclusivity in its partition is achieved by maintaining a continuous LRU stack in both levels.

Due to space limitations, we refer the reader to the description of Karma [30] for details on the structure, generation and transmission of hints, the definition and computation of marginal gain, the management of split partitions, the reorganization of the cache when new hints are supplied by the application, and the applicability of hint based management to various application domains.

Approximate hints. In addition to the looping, sequential, and uniform random accesses handled by Karma, MC^2 also handles *skewed* ranges, which are accessed with non-uniform random distribution. A skew of a range is defined by the pair (x, y), where $x\%$ of the accesses to this range request $y\%$ of its blocks. The skew implies that *some* blocks in the range have a higher access frequency than others, and are thus more valuable, but it does not reveal *which* blocks.

In order to determine what amount of space should be allocated to skewed ranges, previous studies compute the marginal gain of different partition sizes at run time, using ghost caches [15]. In order to avoid the resulting space and computational overheads, we split skewed ranges into several subranges with fixed sizes and decreasing marginal gain. Space is allocated only to the subranges with the highest marginal gain, and their partitions are managed in a continuous LRU stack, allowing LRU to keep the most frequently accessed blocks from this range in the cache.

A skew of (x, y) splits range r into two subranges.The size of the first subrange is $y\%$ of the size of r, and the fre-

quency of access to it is assumed to be $x\%$ of the frequency of r. The second subrange is its complement, both in size and in frequency. The marginal gain is computed for each subrange as if it were accessed uniformly. When the access skew of a data set is available to the application or to an administrator (as in some well-tuned database systems), more accurate hints can be used. A skew with s pairs of parameters, $(x_1, y_1,), ..., (x_s, y_s,)$, divides a range into $s + 1$ subranges, enabling refined allocation decisions. When no information is available, MC^2 follows Pareto's principle and divides the range according to a skew of (80,20).

3.2. Global allocation

We partition the cache between clients according to a global allocation scheme similar to that used in LRU-SP [4] for multiple processes. Initially, each client is allocated an equal share of the cache, which is then dynamically adjusted to reflect this client's use of the shared cache. Whenever a new block is fetched into the cache, a *victim client* is chosen – a client whose block will be evicted to make room for the new block. In global LRU allocation, this client is the one holding the LRU block in the cache. This approach penalizes a client that does not use LRU replacement in its share of the cache. Such a client will not replace the LRU block, and will be constantly chosen as victim. To avoid this problem, we maintain an LRU stack of *partitions*. The victim client is the owner of the *least recently used partition*. The share of the victim client is decreased, and a block from its lowest priority partition is evicted. The allocation for the client whose block entered the cache is increased. Note that when its allocation increases, a client may temporarily save a block with a low marginal gain. The space occupied by this block will later be used to increase the size of a partition with a higher marginal gain.

The result of the global allocation scheme is that MC^2 *favors clients that use the shared cache both frequently and efficiently*. Each access can potentially increase a client's share of the cache. Thus, frequency is rewarded without being hinted or monitored. Furthermore, we need not worry about clients changing their access frequency, or joining or leaving the system. If a client no longer accesses the cache, its partitions become the LRU ones, and their space is gradually allocated to more active clients. Partitions which are accessed frequently stay in the MRU side of the partition stack. Thus, clients which save valuable blocks are rarely chosen as victims, and are rewarded for efficient use of their share of the cache.

Allocation of shared partitions. When a range is accessed by more than one client, MC^2 manages its blocks in a single *shared partition*, rather than distributing its allocation between the clients' shares. All clients accessing a shared partition contribute to its allocation. The maximal space is allocated, from the share of each client, given the marginal gain it computed for the range and the space available to this client. Since clients may disagree on the size of the range as a result of different allocations in the client caches, the total amount of space allocated for the shared partition does not exceed the maximal size requested by the clients.

Each client accesses the shared partition with a different frequency. Thus, a client with a low frequency might exceed its fair allocation when its blocks are accessed frequently by another client. If that happens, the shared partition will not reach the LRU position in the partition stack, and will not be chosen as victim. This may prevent other clients from saving more valuable blocks in the cache. To address this problem, we refine the LRU allocation scheme described above, and use a stack of $(partition, client)$ pairs. When client c accesses partition p, the pair (p, c) moves to the MRU position in the stack. When a block has to be evicted from the cache, the victim client is the client in the *LRU (partition,client) pair*.

Isolating local repartitioning. When $Client_A$ is supplied with a new set of hints which indicate that its access pattern is about to change, it may repartition its cache or resize some of its partitions [30]. This does not affect the space allocated for other clients, unless they share partitions with $Client_A$. Consider partition P, which is shared by $Client_A$ and $Client_B$. As a result of the new hints, the amount of space allocated for P from $Client_A$'s share may decrease. The allocation in the share of $Client_B$ is unchanged. However, if P holds a range with a high marginal gain for $Client_B$, its lower priority partitions may be resized in order to reallocate the buffers removed by $Client_A$.

3.3. Global replacement

The space allocated for each client in a shared cache is managed according to the hints this client receives from the application and forwards to lower levels. Each I/O request from that client is tagged (by the upper level) with the client's ID, along with the original application hint. This identification refers the block to the partition allocated for its range. The same hints are used by the shared cache to choose the best replacement policy for shared partitions.

The global replacement scheme first chooses between exclusive and inclusive management. Exclusive caching greatly improves the performance of multiple levels of cache used by a single client [29, 30]. However, in the presence of multiple clients, exclusive policies may have a negative effect, and may even degrade performance below that of an inclusive policy. MC^2 distinguishes between two types of partitions. In non-shared partitions, used by a single client, exclusivity is maintained by saving demoted

and READ-SAVE-d blocks, and discarding READ blocks. In shared partitions, exclusivity is turned off and READ requests are treated as if they were READ-SAVE. A shared partition stores blocks that were READ, READ-SAVE-d or demoted. The replacement policy in a shared partition is chosen according to the combination of access patterns, using the following guidelines.

Random. When all the clients access the partition in a random pattern, either with uniform or skewed distribution, LRU is used to manage its blocks. When some blocks have higher access frequency than others, a block requested by one client is likely to be requested soon by another client. Similarly, a demoted block might be requested again by the demoting client. Therefore, both read and demoted blocks are treated equally, allowing LRU to keep in the partition those blocks that are indeed reused.

Loop. Several clients iterating over the same loop may advance at different speeds. A block read [demoted] by one client is likely to be accessed by another client *before* it will be accessed again by the client that just read [demoted] it. In order to allow for this benefit of sharing, shared loop partitions are managed by LRU rather than MRU. This type of management keeps all the blocks in the cache for some time, allowing other clients to access them at a low cost.

Random and scan. When a range is accessed randomly by some clients and scanned by others (either sequentially or in a loop), the scans are not allowed to alter the LRU stack of the partition. The scanning clients benefit from the presence of frequently accessed blocks in the shared cache. The random clients are not harmed because polluting scans do not affect the content of the shared partition.

4. Experimental testbed

While MC^2 is designed to work in n levels of cache, we compare it to existing policies on a testbed consisting of two cache levels ($n = 2$). We simulate a client cache attached to each client in the first level and a shared second level cache.

We use the PostgreSQL database [20] as a source of I/O traces and application hints. The query plan generated by its *explain* mechanism is used to supply MC^2 with the access pattern for each range, along with a division of all the blocks into ranges.

We use two benchmarks defined by the Transaction Processing Council: TPCH for decision support, and TPCC for on-line transaction processing (OLTP) [1].

The TPCH traces are used in two different experiments. The first examines two clients sharing a second level cache, each executing a stream of one TPCH query, requesting different values each time. Following a previous study [28], we chose queries 3 and 7. Query 3 is sequential, scanning 3 tables. Subsequent runs of Query 3 result in a looping reference. Query 7 uses an index tree for most of its accesses. The data set of Query 3 is a subset of the data set of Query 7.

In the second experiment with the TPCH benchmark, each of five clients executes a different permutation of the benchmark queries. The permutations correspond to the query sets defined in the benchmark, each consisting of all 22 benchmark queries, in different order.

The TPCC workload consists mainly of index accesses, with a different access skew to each table. Therefore, recency of access is the best indication that a block will be accessed again. We generate ten separate client traces using TPCC-UVa [18], an open-source implementation of the benchmark, and run them concurrently on our simulator.

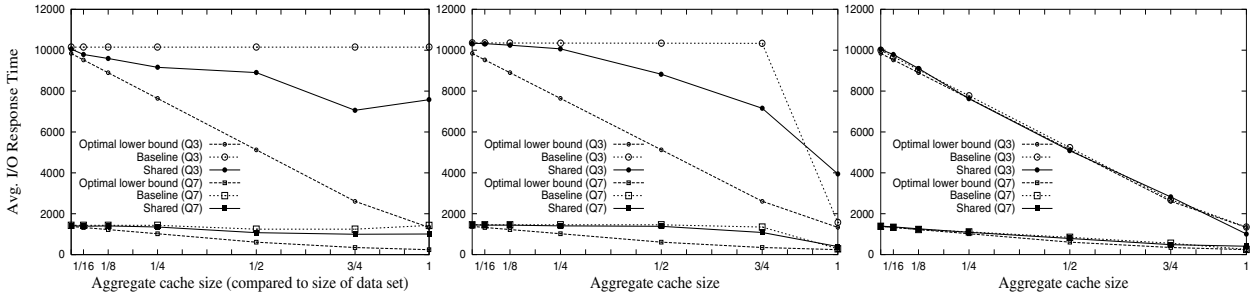
We compared MC^2 to four existing replacement policies: **LRU**, its exclusive version, **Demote** [29], and two policies which account for frequency as well as recency, **ARC** [19] and **MultiQ** [31]. We simulated the above policies, and measured the average I/O response time for each client on a series of increasing cache sizes. We set $C_1 = 1, C_2 = D_2 = 200$ and $C_{Disk} = 10000\mu sec$ [29] and assume, as in previous studies, that all caches are of equal size [29, 30]. We refer the reader to the evaluation of Karma [30] for a discussion of its sensitivity to these system parameters. The cache size in the graphs refers to the aggregate cache size (from the point of view of each client) as a fraction of the size of the data set. Thus, a cache size of 1 means that every single cache is big enough to hold one-half of the data set.

A useful tool for evaluating a cache replacement policy is comparison to the performance of an optimal offline policy. The optimal policy for a single cache is Belady's MIN [3], which evicts the block which will be accessed again furthest in the future. The optimal replacement policy for a multi-level cache in a model with DEMOTE is unknown. Since we are interested in the I/O response time experienced by such a policy, we use Gill's lower bound on the optimal I/O response time for a single client[1] [9].

5. Results

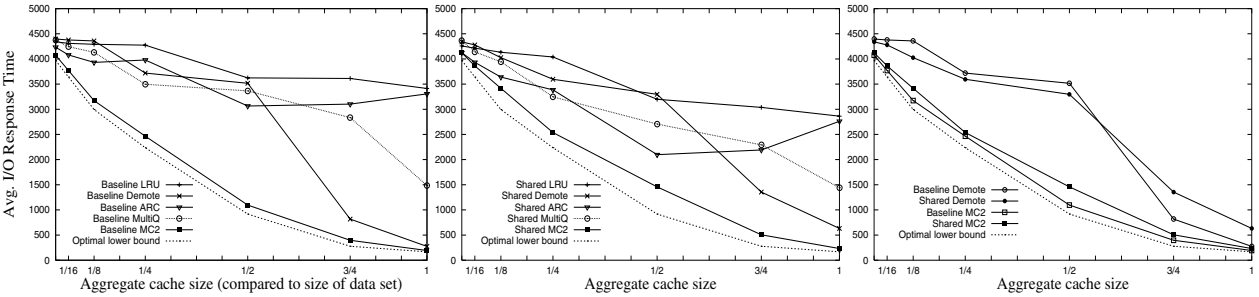
Does MC^2 achieve the shortest I/O response times? Our experimental results show that the answer to this question is yes. Figure 2 shows the results for the pair of TPCH queries, 3 and 7. The performance of MC^2 (Figure 2(c)) is very close to the optimal lower bound, and is the best of all online policies, both for the single client setting and for both clients executing concurrently. Based on the hints, MRU replacement is chosen to manage loop accesses, and LRU is used for index accesses. By combining the right replacement policy with exclusivity, I/O response times decrease linearly with the increase in available cache space.

[1]The extension of this lower bound for multiple clients assumes a fixed interleaving of the clients' requests. We prefer to avoid this assumption in our evaluation and analysis.



(a) ARC (b) Demote (c) MC^2

Figure 2. TPCH queries 3 and 7. Sharing has a positive effect for inclusive policies, or for exclusive policies that demote invaluable blocks. When the performance of a single client ("baseline") is close to the optimal lower bound, sharing has a negative impact.



(a) Average baseline performance (b) Average shared performance (c) Exclusive policies

Figure 3. TPCH query sets on 5 clients. Although the performance of MC^2 degrades due to sharing, its I/O response times are significantly lower than all other online policies, both with a single client ("baseline") and with multiple ones ("shared").

ARC is the best inclusive policy in this experiment. Still, the client running query 3 suffers from managing its loop blocks in an LRU based policy, and the client running query 7 does not utilize all available cache space due to data redundancy. Demote keeps the caches exclusive for the single client workload, but manages the loop blocks with LRU. The situation is somewhat improved by sharing for both policies, but their performance is far from that of MC^2.

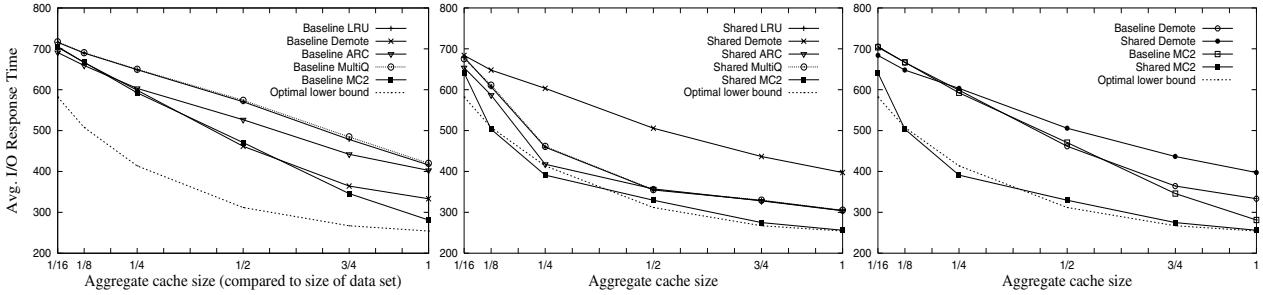
Figure 3 shows the results for the TPCH query sets. MC^2 achieves the shortest I/O response times in both single- and multi-client settings. For a single client, this is the result of combining exclusivity with the best replacement policy for each hinted access pattern. For multiple clients, this is the result of exploiting data sharing whenever it is useful, and partitioning the cache to minimize the interference between clients.

Figure 4 shows the performance of the different policies on 10 TPCC clients. MC^2 performs as well as the exclusive policy in the single client setting. Only in the smallest cache sizes (1/16 and 1/8), demoted blocks are not accessed frequently enough to justify the cost of demotion. MC^2 is better than all inclusive policies for multiple clients, with I/O response times shorter than the lower bound for a single client, thanks to the prefetching effects of sharing.

The hints used by MC^2 in this experiment for the non-uniform accesses were fairly accurate. Random accesses were grouped into a single range, and the skew was computed according to their distribution in the traces. We ran the same experiment using a default (80,20) skew, with similar results. The difference in I/O response time was 0.6% on average for all cache sizes, and was always below 2.1%. This demonstrates that MC^2 is able to save the most valuable blocks in the cache even when supplied with approximate hints.

Does MC^2 benefit from data sharing more than existing policies? Ironically, the answer is no. Since MC^2 performs so well for a single client, it does not leave much room for improvement by sharing. Our results show that the worse performance a policy achieves with a single client, the more it benefits from sharing. This is not surprising; long I/O response times often indicate a poor utilization of the second level cache. This is usually caused by data replication. When several clients share the second level cache, the interleaving of their accesses increases the exclusivity of the cache levels. ARC, MultiQ and LRU demonstrate this behavior in the following experiments.

When the pair of TPCH queries in Figure 2 execute concurrently, loop blocks are accessed out of order. This greatly improves the performance of the inclusive polices, as shown



(a) Average baseline performance (b) Average shared performance (c) Exclusive policies

Figure 4. 10 clients running TPCC. MC^2 maintains exclusive caching for a single client, utilizing the entire aggregate cache. When sharing is identified, exclusivity is disabled in the shared cache. Valuable blocks fetched by one clients act as "prefetches" for another, reducing I/O response times below those of the optimal lower bound for demand paging.

by the results of ARC in Figure 2(a). Figure 4(b) shows results for 10 TPCC clients executing concurrently. Since the access distribution is similar for all clients, the inclusive policies (LRU, MultiQ and ARC) achieve shorter I/O response times than for a single client. Blocks fetched by one client are also used by other clients, so it is useful to keep them in the shared cache.

In some cases, poor utilization of the second level cache is caused by the wrong choice of replacement policy, even when exclusivity is maintained. Specifically, invaluable demoted blocks do not contribute to a reduction in I/O response time. In such cases, when several clients "interfere" with the exclusivity of one another, the blocks saved in the shared cache by one client are useful for another client. This positive effect is demonstrated by Demote in the small cache sizes on TPCH workloads (Figures 2(b) and 3(c)).

MC^2 does benefit most from sharing on the TPCC workload, demonstrating that sharing can also have a positive effect when the cache is utilized well for a single client. When the workload performed by multiple clients is non-uniform, yet similar for all clients, every synchronous fetch by one client is a potential "prefetch" for another. A replacement policy which saves the most "popular" blocks in the cache may achieve shorter wait times than the lower optimal bound for a single client. This positive effect of sharing is demonstrated in Figure 4(c) for MC^2.

MC^2 disables exclusive caching for this workload. Valuable random blocks are kept in the shared cache, to allow for the benefit of data sharing. The looping patterns are identified using hints, and shared loop partitions are managed with LRU in the second level cache. This keeps loop blocks in the cache long enough for other clients to use, creating a prefetching effect. As a result, the I/O response time for multiple clients is shorter than those of the rest of the policies (Figure 4(b)). In some cache sizes they are also lower than the optimal lower bound. This is possible because the lower bound was computed for a single client in a demand paging model.

Policy	LRU	Demote	ARC	MultiQ	MC^2
5 TPCH query sets	16.12%	7.6%	31.54%	19.59%	N/A
TPCH Query 3	29.75%	30.74%	30.44%	31.5%	25.09%
10 TPCC clients	37.83%	30.74%	32.12%	38.07%	34.01%

Table 1. Max. *reduction* in avg. I/O response time due to sharing. N/A means sharing never improves performance.

Clearly, we would like to avoid policies whose benefit from sharing is the result of poor performance for each single client. This appears to be a problem shared by all existing policies. In contrast, MC^2 achieves the best performance for a single client and benefits from sharing by multiple clients. This makes it the best policy for both single- and multi-client workloads. Table 1 summarizes the maximal reduction in I/O response time due to sharing, experienced by all policies.

Is MC^2 better than existing policies in avoiding the negative effects of sharing? In the majority of our experiments, the answer is yes. Sharing has a negative impact on MC^2 only when it cannot be avoided, as opposed to the rest of the policies, which simply make the wrong management decisions.

When the cache hierarchy is fully and efficiently utilized for a single client by a strictly *exclusive* policy, sharing usually has a negative effect. This behavior is observed in two cases. The first is when the aggregate cache is big enough to hold the entire data set or most of it. The I/O response times for an exclusive policy approach those of the optimal lower bound, regardless of its replacement decisions. The impact of sharing in this case is always negative, since it interferes with exclusivity. Demote exhibits such behavior in the large cache sizes for the TPCH traces, in Figures 2(b) and 3(c).

The second case in which sharing and exclusivity create a negative effect is when exclusivity is combined with the best replacement policy for the access pattern. In such cases, clients "steal" valuable blocks from one another, as READ blocks are automatically discarded. Demote demon-

Policy	LRU	Demote	ARC	MultiQ	MC^2
5 TPCH query sets	N/A	130%	N/A	0.21%	32.93%
TPCH Query 3	N/A	151%	N/A	N/A	4.97%
10 TPCC clients	N/A	19.83%	N/A	N/A	N/A

Table 2. Max. *increase* in average I/O response time due to sharing. N/A means sharing never degrades performance.

strates this scenario on the TPCC workload in Figure 4(c).

Exclusivity aside, sharing can also have a negative effect due to competition between clients. When each client is able to fully utilize the shared cache, other clients take up valuable space, harming each other's performance. This is the case of MC^2 in the TPCH query sets in Figure 3(c). The clients execute the queries in different order, and their working sets do not completely overlap. Sharing has a negative impact due to increased load on the shared cache.

Clearly, we would like to avoid exclusive caching whenever it degrades performance. In such cases the negative effect of sharing indicates poor management, as it does in the case of Demote. In contrast, a negative impact which results purely from increased load on the shared resources cannot be avoided, and does not indicate a weakness of the management policy. This is the case of MC^2, which achieves the best performance both for a single client and for multiple ones. Table 2 summarizes the maximal increase in I/O response time due to sharing, experienced by all policies.

We conclude that MC^2 is the best policy for managing a hierarchy of multiple cache levels used by multiple clients. it achieves the shortest I/O response times in all our experiments. By combining exclusive caching and application hints it guarantees the best performance for a single client workload. The same principles are used for each client in a multi-client setting, by the local allocation and replacement scheme. The application hints are further leveraged to choose between inclusive and exclusive caching, optimizing performance in the presence of multiple clients. Thanks to the global allocation scheme, the most valuable blocks in the system are saved in the shared cache, while maintaining a high degree of fairness between competing clients.

6. Related work

Several approaches are used to handle caching in the presence of data sharing.

Policies oblivious to sharing. Many replacement policies are oblivious to the cache level they manage and to the number of clients accessing the cache. Their performance may be implicitly affected by the interleaving of access patterns, but no effort is made to distinguish between different sources of requests. Examples for such policies are LRU, MRU, ARC [19], and MultiQ [31]. Some policies which attempt to optimize aspects of disk scheduling, such as DiskSeen [7] and STEP [17], also fall into this category.

Exclusive multilevel policies. Exclusive caching is difficult to achieve and is not always efficient in a system with multiple clients. Several policies designed for global management of the cache hierarchy do not address the problems demonstrated in our experiments. Some examples are the algorithm derived from using write hints [16], X-Ray [26], the global algorithm in [31], and Karma [30]. In *heterogeneous caching* [2], some degree of exclusivity is achieved by managing each cache level with a different policy.

A different group of multi-level cache policies specifically address multiple clients. The first to do so was Demote [29], when the notion of exclusive caching was first introduced. Ghost caches are used to estimate the value of demoted blocks compared to that of READ ones. PROMOTE [9] uses probabilistic filtering to decide which blocks are forwarded to upper cache levels. A block forwarded to one client is removed from the shared cache. In ULC [14], the client instructs the server which blocks to save in its cache. Global LRU allocation is employed for multiple clients. The issue of shared data is not addressed. In uCache [22] a small LRU cache is allocated for high-correlated data, and the server keeps track of blocks saved in each client.

MC^2 does not assume, or detect, the degree of sharing. Instead, shared blocks are identified by client hints, which also disclose the combination of access patterns. Exclusive caching is maintained only when it is known to be useful.

Cooperation. Cooperative caching in distributed file systems can significantly reduce the amount of disk accesses. Most protocols incur varying amounts of overhead of global lookup and management [6, 22, 25, 27]. When clients leave or join the system its state is altered, compromising the efficiency of the protocol.

MC^2 achieves very low I/O response times by taking advantage of sharing in the shared cache, with less overhead and vulnerability to change.

Cache partitioning. Partitioning is used widely in policies designed for multiple processes sharing the file system cache. One approach partitions the cache between detected access patterns [12, 15]. Allocation is based on the observed marginal gain of each partition. The detection mechanisms assume that each block is accessed in one access pattern. Argon [28] uses a similar approach for a storage server. Its services are assumed to be completely independent with no data sharing. Several policies designed to manage the buffer cache of a database server specifically address data sharing. The owner is either the first query to fetch the block [24], or the transaction which assigns it the highest priority [13].

Prefetching as a form of data sharing. Although prefetching is not part of our storage model, its effects may be similar to those of sharing. Prefetched blocks are accessed in two interleaving patterns: the one in which they are read by the application, and the sequential access of

prefetching. Even when the application accesses the blocks sequentially, its accesses are not necessarily synchronized with the stream of prefetches.

SARC [11] and AMP [10] combine caching and prefetching in a storage server. SARC adjusts the stack positions of prefetched blocks to avoid eviction. In AMP the degree of prefetching is adapted separately for each stream. When prefetching is done by the application itself, its management can be combined with additional information disclosed by the application, as in TIP2 [23] and LRU-SP [4].

Although MC^2 does not address prefetching explicitly, it enables clients to enjoy effects similar to prefetching, when clients use each other's blocks. By addressing different sharing scenarios, MC^2 lays the foundations for combining caching and prefetching in a positive manner.

7. Conclusions

MC^2 leverages the same hints for choosing the best replacement and allocation schemes for a single client's access pattern, and the *combination* of access patterns of multiple clients. The hints are also used to derive the degree and type of data sharing, and choose between inclusive or exclusive caching accordingly. Our experiments show that the I/O response times achieved by MC^2 are significantly lower than those achieved by existing policies.

References

[1] http://www.tpc.org/information/benchmarks.asp.

[2] I. Ari. *Design and Management of Globally Distributed Network Caches*. PhD thesis, UC Santa Cruz, 2004.

[3] L. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM TOCS*, 14(4):311–343, 1996.

[5] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB*, 1985.

[6] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *OSDI*, 1994.

[7] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *USENIX Annual Tech.*, 2007.

[8] R. Gabor, S. Weiss, and A. Mendelson. Fairness enforcement in switch on event multithreading. *ACM Trans. Archit. Code Optim.*, 4(3):15, 2007.

[9] B. Gill. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *FAST*, 2008.

[10] B. S. Gill and L. A. D. Bathen. AMP: adaptive multi-stream prefetching in a shared cache. In *FAST*, 2007.

[11] B. S. Gill and D. S. Modha. SARC: Sequential prefetching in adaptive replacement cache. In *USENIX Annual Tech.*, 2005.

[12] C. Gniady, A. R. Butt, and Y. C. Hu. Program counter based pattern classification in buffer caching. In *OSDI*, 2004.

[13] R. Jauhari, M. J. Carey, and M. Livny. Priority-hints: An algorithm for priority-based buffer management. In *VLDB*, 1990.

[14] S. Jiang and X. Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *ICDCS*, 2004.

[15] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *OSDI*, 2000.

[16] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao. Second-tier cache management using write hints. In *FAST*, 2005.

[17] S. Liang, S. Jiang, and X. Zhang. STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *ICDCS*, 2007.

[18] D. R. Llanos and B. Palop. An open-source TPC-C implementation for parallel and distributed systems. In *IPDPS*, 2006.

[19] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, 2003.

[20] B. Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley, 2000.

[21] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *SIGMOD*, 1991.

[22] L. Ou, X. He, M. J. Kosa, and S. L. Scott. A unified multiple-level cache for high performance storage systems. In *MASCOTS*, 2005.

[23] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In H. Jin, T. Cortes, and R. Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 224–244. IEEE Computer Society Press and Wiley, NY, 2001.

[24] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM TODS*, 11(4), 1986.

[25] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *OSDI*, 1996.

[26] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *FAST*, 2005.

[27] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin, and H. M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *SIGMETRICS*, 1998.

[28] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *FAST*, 2007.

[29] T. M. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *USENIX Annual Tech.*, 2002.

[30] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-all replacement for a multilevel cache. In *FAST*, 2007.

[31] Y. Zhou, Z. Chen, and K. Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, 2004.